Scientific software engineering for a simple ODE model

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Aug 18, 2014

Contents

Goal.

This document illustrates *best practice* for developing scientific softwa an efficient and reliable way. Not only will the outlined techniques a lost of human time, but they will also help assure reproducible sc and higher quality of computational investigations. Key questions answered are

- How should I organize a program?
- How can I efficiently and safely provide input data and run my
- How can I verify that the implementation is correct?
- How should I reliably work with files and documents?
- How should I conduct large numerical experiments?

hpl 1: Need to cover functions, classes, modules, cml, GUI, ha trivial problems, exact num sol, MMS, qualitative results, Git, bitbucket scripting, report generation

1 Sample problem and code

This first introduction to good programming habits in scientific comput make use of a very simple mathematical problem to keep the mathe details at the lowest possible level while introducing a series of computer concepts. The simplicity of the mathematical problem obviously prev from treating several techniques that are only meaningful for complex s software.

.1 Mathematical problem

We consider the simplest possible ordinary differential equation with constant coefficient a:

$$u'(t) = -au(t), \quad u(0) = I, \quad t \in (0, T]\,. \tag{1}$$

This problem is numerically solved by the so-called θ-rule, which is a convenient way to merge different formulas for the well-known Forward Euler, Backward Euler, and Crank-Nicolson (midpoint/central) schemes. We introduce a uniform time mesh $t_n = n\Delta t$, $n = 0, 1, \ldots, N_t$, and seek $u(t)$ at the mesh points. The numerical approximation to $u(t_n)$ is denoted u^n. Since we will use the symbol u both for the exact analytical solution of (1) and for the numerical approximation, we sometimes introduce $u_{\mathrm{e}}(t)$ to help distinguish the two types of solutions (i.e., subscript e for "exact")[1].

The θ-rule leads to an explicit updating formula for u^{n+1}, given u^n:

$$u^{n+1} = \frac{1 - (1-\theta)a\Delta t}{1 + \theta a\Delta t} u^n,$$

.2 Implementation

The numerical method is implemented as a function `solver`. Another function `explore` computes the error in the solution, by comparing with the exact solution $u_{\mathrm{e}}(t) = Ie^{-at}$, and creates a plot for comparing the numerical and exact solution.

The program file `decay_plot.py`[2] contains the two functions and a main program.

```
from numpy import *
from matplotlib.pyplot import *

def solver(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    dt = float(dt)            # avoid integer division
    Nt = int(round(T/dt))     # no of time intervals
    T = Nt*dt                 # adjust T to fit time step dt
    u = zeros(Nt+1)           # array of u[n] values
    t = linspace(0, T, Nt+1)  # time mesh

    u[0] = I                  # assign initial condition
    for n in range(0, Nt):    # n=0,1,...,Nt-1
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t

def exact_solution(t, I, a):
    return I*exp(-a*t)
```

[1] In the literature, it is more common to put a subscript (like u_Δ or u_h) on the numerical solution to distinguish it from the exact solution. However, we will use the variable `u` in the code for the numerical approximation to be computed, and therefore adjust the mathematical notation to convenient conventions in the code such that we can have as close correspondence as possible between the implementation and the mathematics.

[2] `http://tinyurl.com/jvzzcfn/softeng1/decay_plot.py`


```
def explore(I, a, T, dt, theta=0.5, makeplot=True):
    """
    Run a case with the solver, compute error measure,
    and plot the numerical and exact solutions (if makeplot=True
    """
    u, t = solver(I, a, T, dt, theta)    # Numerical solution
    u_e = exact_solution(t, I, a)
    e = u_e - u
    E = sqrt(dt*sum(e**2))
    if makeplot:
        figure()                         # create new plot
        t_e = linspace(0, T, 1001)       # fine mesh for u_e
        u_e = exact_solution(t_e, I, a)
        plot(t,   u,   'r--o')           # red dashes w/circles
        plot(t_e, u_e, 'b-')             # blue line for exact s
        legend(['numerical', 'exact'])
        xlabel('t')
        ylabel('u')
        title('theta=%g, dt=%g' % (theta, dt))
        theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
        savefig('%s_%g.png' % (theta2name[theta], dt))
        savefig('%s_%g.pdf' % (theta2name[theta], dt))
        show()
    return E

def main(I, a, T, dt_values, theta_values=(0, 0.5, 1)):
    for theta in theta_values:
        for dt in dt_values:
            E = explore(I, a, T, dt, theta, makeplot=True)
            print '%3.1f %6.2f: %12.3E' % (theta, dt, E)

main(I=1, a=2, T=5, dt_values=[0.4, 0.04])
```

2 User interfaces

It is good programming practice to let programs read input from the use than require the user to edit the source code when trying out new values parameters. One reason is that any edit of the code has a danger of intr bugs. Another reason is that it is easier and less manual work to supply a program instead of editing the program code. A third reason is that a p that reads input can easily be run by another program, and in this way automate a large number of runs in scientific investigations.

Tip.

We shall make it a habit to equip any implementation of a numerical s with an appropriate user interface before testing out the code.

Reading input data can be done in many ways. We have to decide on *user interface*, i.e., how we want to operate the program when providin and then use appropriate tools to implement the user interface. Th

our basic types of user interface of relevance to our programs, listed here with ncreasing complexity of the implementation:

1. Questions and answers in the terminal window
2. Command-line arguments
3. Reading data from file
4. Graphical user interfaces

lthough conceptually simple, alternative 1 involves more typing than the other lternatives and is therefore abandoned. Below, we shall address alternative 2 nd 4, which are most appropriate for the present problem.

[[[

.1 Creating command-line interfaces

eading input from the command line is a simple and flexible way of interact- ng with the user. Python stores all the command-line arguments in the list `ys.argv`, and there are, in principle, two ways of programming with command- ne arguments in Python:

- Decide upon a sequence of parameters on the command line and read their values directly from the `sys.argv[1:]` list (`sys.argv[0]` is the just program name).
- Use option-value pairs (`-option value`) on the command line to override default values of input parameters, and utilize the `argparse.ArgumentParser` tool to interact with the command line.

oth strategies will be illustrated next.

eading a sequence of command-line arguments. The `decay_plot.py`[3] rogram needs the following input data: I, a, T, an option to turn the plot on r off (`makeplot`), and a list of Δt values.

The simplest way of reading this input from the command line is to say that ne first four command-line arguments correspond to the first four points in the st above, in that order, and that the rest of the command-line arguments are ne Δt values. The input given for `makeplot` can be a string among `'on'`, `'off'`, `True'`, and `'False'`. The code for reading this input is most conveniently put n a function:

[3] `http://tinyurl.com/jvzzcfn/softeng1/decay_plot.py`


```
import sys

def read_command_line():
    if len(sys.argv) < 6:
        print 'Usage: %s I a T on/off dt1 dt2 dt3 ...' % \
              sys.argv[0]; sys.exit(1)  # abort

    I = float(sys.argv[1])
    a = float(sys.argv[2])
    T = float(sys.argv[3])
    makeplot = sys.argv[4] in ('on', 'True')
    dt_values = [float(arg) for arg in sys.argv[5:]]

    return I, a, T, makeplot, dt_values
```

One should note the following about the constructions in the prograr

- Everything on the command line ends up in a *string* in the list `sy` Explicit conversion to, e.g., a `float` object is required if the stri number we want to compute with.
- The value of `makeplot` is determined from a boolean expression becomes `True` if the command-line argument is either `'on'` or `'Tru` `False` otherwise.
- It is easy to build the list of Δt values: we simply run through of the list, `sys.argv[5:]`, convert each command-line argument t and collect these `float` objects in a list, using the compact and co *list comprehension* syntax in Python.

The loops over θ and Δt values can be coded in a `main` function:

```
def main():
    I, a, T, makeplot, dt_values = read_command_line()
    for theta in 0, 0.5, 1:
        for dt in dt_values:
            E = explore(I, a, T, dt, theta, makeplot)
            print '%3.1f %6.2f: %12.3E' % (theta, dt, E)
```

The complete program can be found in `decay_cml.py`[4].

Working with an argument parser. Python's `ArgumentParser` to `argparse` module makes it easy to create a professional command-line i to any program. The documentation of `ArgumentParser`[5] demonstr versatile applications, so we shall here just list an example containi features. On the command line we want to specify option-value pairs and T, e.g., `-a 3.5 -I 2 -T 2`. Including `-makeplot` turns the plot excluding this option turns the plot off. The Δt values can be given a `0.5 0.25 0.1 0.01`. Each parameter must have a sensible default valu

[4] `http://tinyurl.com/jvzzcfn/softeng1/decay_cml.py`
[5] `http://docs.python.org/library/argparse.html`

e specify the option on the command line only when the default value is not uitable.

We introduce a function for defining the mentioned command-line options:

```
lef define_command_line_options():
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--I', '--initial_condition', type=float,
                        default=1.0, help='initial condition, u(0)',
                        metavar='I')
    parser.add_argument('--a', type=float,
                        default=1.0, help='coefficient in ODE',
                        metavar='a')
    parser.add_argument('--T', '--stop_time', type=float,
                        default=1.0, help='end time of simulation',
                        metavar='T')
    parser.add_argument('--makeplot', action='store_true',
                        help='display plot or not')
    parser.add_argument('--dt', '--time_step_values', type=float,
                        default=[1.0], help='time step values',
                        metavar='dt', nargs='+', dest='dt_values')
    return parser
```

Each command-line option is defined through the `parser.add_argument` ıethod. Alternative options, like the short `-I` and the more explaining version `-initial_condition` can be defined. Other arguments are `type` for the Python bject type, a default value, and a help string, which gets printed if the command-ne argument `-h` or `-help` is included. The `metavar` argument specifies the alue associated with the option when the help string is printed. For example, ıe option for I has this help output:

```
erminal> python decay_argparse.py -h
 ...
 --I I, --initial_condition I
                       initial condition, u(0)
 ...
```

he structure of this output is

```
    --I metavar, --initial_condition metavar
                          help-string
```

The `-makeplot` option is a pure flag without any value, implying a true value the flag is present and otherwise a false value. The `action='store_true'` ıakes an option for such a flag.

Finally, the `-dt` option demonstrates how to allow for more than one value eparated by blanks) through the `nargs='+'` keyword argument. After the ɔmmand line is parsed, we get an object where the values of the options are ɔred as attributes. The attribute name is specified by the `dist` keyword rgument, which for the `-dt` option is `dt_values`. Without the `dest` argument, ıe value of an option `-opt` is stored as the attribute `opt`.

The code below demonstrates how to read the command line and ext values for each option:

```
def read_command_line():
    parser = define_command_line_options()
    args = parser.parse_args()
    print 'I={}, a={}, T={}, makeplot={}, dt_values={}'.format(
        args.I, args.a, args.T, args.makeplot, args.dt_values)
    return args.I, args.a, args.T, args.makeplot, args.dt_values
```

The `main` function remains the same as in the `decay_cml.py` code b reading from `sys.argv` directly. A complete program featuring the dem of `ArgumentParser` appears in the file `decay_argparse.py`[6].

2.2 Creating a graphical web user interface

The Python package Parampool[7] can be used to automatically generat based *graphical user interface* (GUI) for our simulation program. A the programming technique dramatically simplifies the efforts to create the forthcoming material on equipping our `decay_mod` module with a quite technical and of significantly less importance than knowing how to command-line interface (Section 2.1). There is no danger in jumping Section 3.4.

Making a compute function. The first step is to identify a functi performs the computations and that takes the necessary input varia arguments. This is called the *compute function* in Parampool terminol may start with a copy of the basic file `decay_plot.py`[8], which has function displayed in Section **??** for carrying out simulations and plot a series of Δt values. Now we want to control and view the same expe from a web GUI.

To tell Parampool what type of input data we have, we assign defaul of the right type to all arguments in the main function and call it `main`

```
def main_GUI(I=1.0, a=.2, T=4.0,
             dt_values=[1.25, 0.75, 0.5, 0.1],
             theta_values=[0, 0.5, 1]):
```

The compute function must return the HTML code we want for dis the result in a web page. Here we want to show plots of the numerical ar solution for different methods and Δt values. The plots can be organi table with θ (methods) varying through the columns and Δt varying thro rows. Assume now that a new version of the `explore` function not only the error `E` but also HTML code containing the plot. Then we can w `main_GUI` function as

[6] `http://tinyurl.com/jvzzcfn/softeng1/decay_argparse.py`
[7] `https://github.com/hplgit/parampool`
[8] `http://tinyurl.com/jvzzcfn/softeng1/decay_plot.py`

```
def main_GUI(I=1.0, a=.2, T=4.0,
         dt_values=[1.25, 0.75, 0.5, 0.1],
         theta_values=[0, 0.5, 1]):
    # Build HTML code for web page. Arrange plots in columns
    # corresponding to the theta values, with dt down the rows
    theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
    html_text = '<table>\n'
    for dt in dt_values:
        html_text += '<tr>\n'
        for theta in theta_values:
            E, html = explore(I, a, T, dt, theta, makeplot=True)
            html_text += """
<td>
<center><b>%s, dt=%g, error: %s</b></center><br>
%s
</td>
""" % (theta2name[theta], dt, E, html)
        html_text += '</tr>\n'
    html_text += '</table>\n'
    return html_text
```

Rather than creating plot files and showing the plot on the screen, the new ersion of the `explore` function makes a string with the PNG code of the plot nd embeds that string in HTML code. This action is conveniently performed y Parampool's `save_png_to_str` function:

```
import matplotlib.pyplot as plt
...
# plot
plt.plot(t, u, 'r-')
plt.xlabel('t')
plt.ylabel('u')
...
from parampool.utils import save_png_to_str
html_text = save_png_to_str(plt, plotwidth=400)
```

ote that we now write `plt.plot`, `plt.xlabel`, etc. The `html_text` string is ng and contains all the characters that build up the PNG file of the current lot. The new `explore` function can make use of the above code snippet and eturn `html_text` along with `E`.

enerating the user interface. The web GUI is automatically generated y the following code, placed in a file `decay_GUI_generate.py`[9]

```
from parampool.generator.flask import generate
from decay_GUI import main
generate(main,
         output_controller='decay_GUI_controller.py',
         output_template='decay_GUI_view.py',
         output_model='decay_GUI_model.py')
```

unning the `decay_GUI_generate.py` program results in three new files whose ames are specified in the call to `generate`:

[9] `http://tinyurl.com/jvzzcfn/softeng1/decay_GUI_generate.py`

1. `decay_GUI_model.py` defines HTML widgets to be used to set inp in the web interface,
2. `templates/decay_GUI_views.py` defines the layout of the web pa
3. `decay_GUI_controller.py` runs the web application.

We only need to run the last program, and there is no need to look in files.

Running the web application. The web GUI is started by

```
Terminal> python decay_GUI_controller.py
```

Open a web browser at the location `127.0.0.1:5000`. Input fields for `dt_values`, and `theta_values` are presented. Setting the latter two to `0.5]` and `[1, 0.5]`, respectively, and pressing *Compute* results in fou see Figure 1. With the techniques demonstrated here, one can easily tailored web GUI for a particular type of application and use it to inter explore physical and numerical effects.

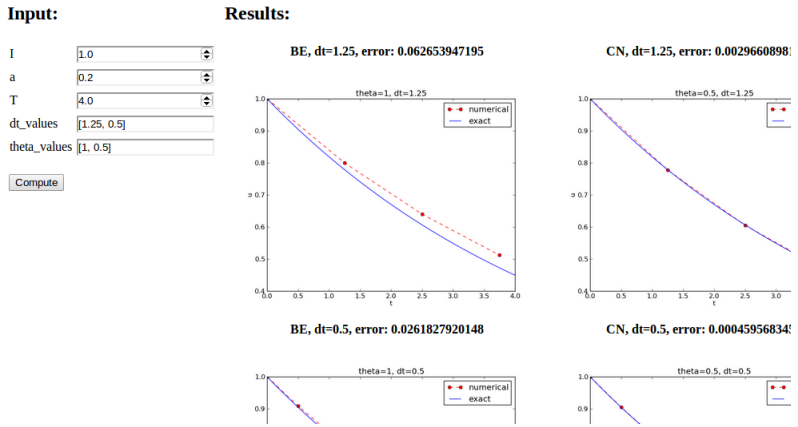


Figure 1: Automatically generated graphical web interface.

3 Verification

3.1 Comparison with hand calculations

One of the simplest and most powerful methods for verifying numeric is to perform some steps of the algorithm by hand and compare the

ith those produced by the code. In the present case, we may choose some test roblem and run three steps by hand. Picking $a(t) = t^2$... **hpl 2**: Not ready. ime-dep a?

.2 Test function

Caution: choice of parameter values.

For the choice of values of parameters in verification tests one should stay away from integers, especially 0 and 1, as these can simplify formulas too much for test purposes. For example, with $\theta = 1$ the nominator in the formula for u^n will be the same for all a and Δt values. One should therefore choose more "arbitrary" values, say $\theta = 0.8$ and $I = 0.1$.

.3 Comparison with an exact discrete solution

ometimes it is possible to find a closed-form *exact discrete solution* that fulfills ıe discrete finite difference equations. The implementation can then be verified gainst the exact discrete solution. This is usually the best technique for erification.

Define

$$A = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}.$$

Ianual computations with the θ-rule results in

$$\begin{aligned} u^0 &= I, \\ u^1 &= Au^0 = AI, \\ u^2 &= Au^1 = A^2 I, \\ &\vdots \\ u^n &= A^n u^{n-1} = A^n I\,. \end{aligned}$$

/e have then established the exact discrete solution as

$$u^n = IA^n\,. \tag{2}$$

Caution.

One should be conscious about the different meanings of the notation on the left- and right-hand side of (2): on the left, n in u^n is a superscript reflecting a counter of mesh points (t_n), while on the right, n is the power in the exponentiation A^n.

Comparison of the exact discrete solution and the computed solution in the following function:

```
def verify_exact_discrete_solution():

    def exact_discrete_solution(n, I, a, theta, dt):
        A = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)
        return I*A**n

    theta = 0.8; a = 2; I = 0.1; dt = 0.8
    Nt = int(8/dt)  # no of steps
    u, t = solver(I=I, a=a, T=Nt*dt, dt=dt, theta=theta)
    u_de = array([exact_discrete_solution(n, I, a, theta, dt)
                  for n in range(Nt+1)])
    difference = abs(u_de - u).max()  # max deviation
    tol = 1E-15  # tolerance for comparing floats
    success = difference <= tol
    return success
```

The complete program is found in the file `decay_verf2.py`[10] (`verf2` is name for "verification, version 2").

Local functions.

One can define a function inside another function, here called a *function* (also known as *closure*) inside a *parent function*. A local tion is invisible outside the parent function. A convenient proper that any local function has access to all variables defined in the p function, also if we send the local function to some other functic argument (!). In the present example, it means that the local fun `exact_discrete_solution` does not need its five arguments as the v can alternatively be accessed through the local variables defined i parent function `verify_exact_discrete_solution`. We can send su `exact_discrete_solution` without arguments to any other functior `exact_discrete_solution` will still have access to `n`, `I`, `a`, and so defined in its parent function.

3.4 Computing convergence rates

We expect that the error E in the numerical solution is reduced if the m Δt is decreased. More specifically, many numerical methods obey a po relation between E and Δt:

$$E = C\Delta t^r,$$

where C and r are (usually unknown) constants independent of Δt. The (3) is viewed as an asymptotic model valid for sufficiently small Δt. Ho is normally hard to estimate without doing numerical estimations of r.

[10] `http://tinyurl.com/jvzzcfn/softeng1/decay_verf2.py`

The parameter r is known as the *convergence rate*. For example, if the convergence rate is 2, halving Δt reduces the error by a factor of 4. Diminishing Δt then has a greater impact on the error compared with methods that have $r = 1$. For a given value of r, we refer to the method as of r-th order. First- and second-order methods are most common in scientific computing.

Estimating r. There are two alternative ways of estimating C and r based on a set of m simulations with corresponding pairs $(\Delta t_i, E_i)$, $i = 0, \ldots, m-1$, and $\Delta t_i < \Delta t_{i-1}$ (i.e., decreasing cell size).

1. Take the logarithm of (3), $\ln E = r \ln \Delta t + \ln C$, and fit a straight line to the data points $(\Delta t_i, E_i)$, $i = 0, \ldots, m-1$.
2. Consider two consecutive experiments, $(\Delta t_i, E_i)$ and $(\Delta t_{i-1}, E_{i-1})$. Dividing the equation $E_{i-1} = C\Delta t_{i-1}^r$ by $E_i = C\Delta t_i^r$ and solving for r yields

$$r_{i-1} = \frac{\ln(E_{i-1}/E_i)}{\ln(\Delta t_{i-1}/\Delta t_i)} \tag{4}$$

for $i = 1, \ldots, m-1$.

The disadvantage of method 1 is that (3) might not be valid for the coarsest meshes (largest Δt values). Fitting a line to all the data points is then misleading. Method 2 computes convergence rates for pairs of experiments and allows us to see if the sequence r_i converges to some value as $i \to m-2$. The final r_{m-2} can then be taken as the convergence rate. If the coarsest meshes have a differing rate, the corresponding time steps are probably too large for (3) to be valid. That is, those time steps lie outside the asymptotic range of Δt values where the error behaves like (3).

Implementation. It is straightforward to extend the `main` function in the program `decay_argparse.py` with statements for computing $r_0, r_1, \ldots, r_{m-2}$ from (3):

```
from math import log

def main():
    I, a, T, makeplot, dt_values = read_command_line()
    r = {}  # estimated convergence rates
    for theta in 0, 0.5, 1:
        E_values = []
        for dt in dt_values:
            E = explore(I, a, T, dt, theta, makeplot=False)
            E_values.append(E)

        # Compute convergence rates
        m = len(dt_values)
        r[theta] = [log(E_values[i-1]/E_values[i])/
                    log(dt_values[i-1]/dt_values[i])
                    for i in range(1, m, 1)]
    for theta in r:
        print '\nPairwise convergence rates for theta=%g:' % thet
        print ' '.join(['%.2f' % r_ for r_ in r[theta]])
    return r
```

The program containing this `main` function is called `decay_convrate.p`

The `r` object is a *dictionary of lists*. The keys in this dictionary a values. For example, `r[1]` holds the list of the r_i values corresponding t In the loop `for theta in r`, the loop variable `theta` takes on the valu keys in the dictionary `r` (in an undetermined ordering). We could sim `print r[theta]` inside the loop, but this would typically yield outpu convergence rates with 16 decimals:

```
[1.331919482274763, 1.1488178494691532, ...]
```

Instead, we format each number with 2 decimals, using a list compre to turn the list of numbers, `r[theta]`, into a list of formatted strings. T join these strings with a space in between to get a sequence of rates on in the terminal window. More generally, `d.join(list)` joins the string list `list` to one string, with `d` as delimiter between `list[0]`, `list[1]`,

Here is an example on the outcome of the convergence rate comput

```
Terminal> python decay_convrate.py --dt 0.5 0.25 0.1 0.05 0.025 0.
...
Pairwise convergence rates for theta=0:
1.33 1.15 1.07 1.03 1.02

Pairwise convergence rates for theta=0.5:
2.14 2.07 2.03 2.01 2.01

Pairwise convergence rates for theta=1:
0.98 0.99 0.99 1.00 1.00
```

The Forward and Backward Euler methods seem to have an r valu stabilizes at 1, while the Crank-Nicolson seems to be a second-order with $r = 2$.

Very often, we have some theory that predicts what r is for a nu method. Various theoretical error measures for the θ-rule point to r $\theta = 0.5$ and $r = 1$ otherwise. The computed estimates of r are in ve agreement with these theoretical values.

Why convergence rates are important.

The strong practical application of computing convergence rates i verification: wrong convergence rates point to errors in the code, correct convergence rates brings evidence that the implementation i

[11] `http://tinyurl.com/jvzzcfn/softeng1/decay_convrate.py`

rect. Experience shows that bugs in the code easily destroy the expected convergence rate.

)ebugging via convergence rates. Let us experiment with bugs and see ıe implication on the convergence rate. We may, for instance, forget to multiply y `a` in the denominator in the updating formula for `u[n+1]`:

```
ı[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt)*u[n]
```

unning the same `decay_convrate.py` command as above gives the expected ɔnvergence rates (!). Why? The reason is that we just specified the Δt values re relied on default values for other parameters. The default value of a is 1. orgetting the factor `a` has then no effect. This example shows how important it to avoid parameters that are 1 or 0 when verifying implementations. Running ıe code `decay_v0.py` with $a = 2.1$ and $I = 0.1$ yields

```
erminal> python decay_convrate.py --a 2.1 --I 0.1  \
         --dt 0.5 0.25 0.1 0.05 0.025 0.01
..
airwise convergence rates for theta=0:
.49 1.18 1.07 1.04 1.02

airwise convergence rates for theta=0.5:
1.42 -0.22 -0.07 -0.03 -0.01

airwise convergence rates for theta=1:
.21 0.12 0.06 0.03 0.01
```

his time we see that the expected convergence rates for the Crank-Nicolson nd Backward Euler methods are not obtained, while $r = 1$ for the Forward uler method. The reason for correct rate in the latter case is that $\theta = 0$ and ıe wrong `theta*dt` term in the denominator vanishes anyway.

The error

```
ı[n+1] = ((1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
```

ıanifests itself through wrong rates $r \approx 0$ for all three methods. About the ıme results arise from an erroneous initial condition, `u[0] = 1`, or wrong loop mits, `range(1,Nt)`. It seems that in this simple problem, most bugs we can ıink of are detected by the convergence rate test, provided the values of the ıput data do not hide the bug.

A `verify_convergence_rate` function could compute the dictionary of list ła `main` and check if the final rate estimates (r_{m-2}) are sufficiently close to the xpected ones. A tolerance of 0.1 seems appropriate, given the uncertainty in stimating r:


```
def verify_convergence_rate():
    r = main()
    tol = 0.1
    expected_rates = {0: 1, 1: 1, 0.5: 2}
    for theta in r:
        r_final = r[theta][-1]
        diff = abs(expected_rates[theta] - r_final)
        if diff > tol:
            return False
    return True  # all tests passed
```

We remark that `r[theta]` is a list and the last element in any list extracted by the index `-1`.

4 Software engineering

Goal.

Efficient use of differential equation models requires software that is to test and flexible for setting up extensive numerical experiments. section introduces three important concepts:

- Modules
- Testing frameworks
- Implementation with classes

The concepts are introduced using the differential equation problem $-au$, $u(0) = I$, as example.

4.1 Making a module

The DRY principle.

The previous sections have outlined numerous different programs, them having their own copy of the `solver` function. Such copies o same piece of code is against the important *Don't Repeat Yourself* (I principle in programming. If we want to change the `solver` function should be *one and only one* place where the change needs to be perforı

To clean up the repetitive code snippets scattered among the `deca` files, we start by collecting the various functions we want to keep for th in one file, now called `decay_mod.py`[12] (`mod` stands for "module"). The fı functions are copied to this file:

[12] `http://tinyurl.com/jvzzcfn/softeng1/decay_mod.py`

- `solver` for computing the numerical solution
- `verify_three_steps` for verifying the first three solution points against hand calculations
- `verify_discrete_solution` for verifying the entire computed solution against an exact formula for the numerical solution
- `explore` for computing and plotting the solution
- `define_command_line_options` for defining option-value pairs on the command line
- `read_command_line` for reading input from the command line, now extended to work both with `sys.argv` directly and with an `ArgumentParser` object
- `main` for running experiments with $\theta = 0, 0.5, 1$ and a series of Δt values, and computing convergence rates
- `main_GUI` for doing the same as the `main` function, but modified for automatic GUI generation
- `verify_convergence_rate` for verifying the computed convergence rates against the theoretically expected values

We use Matplotlib for plotting. A sketch of the `decay_mod.py` file, with complete versions of the modified functions, looks as follows:

```
from numpy import *
from matplotlib.pyplot import *
import sys

def solver(I, a, T, dt, theta):
    ...

def verify_three_steps():
    ...

def verify_exact_discrete_solution():
    ...

def u_exact(t, I, a):
    ...

def explore(I, a, T, dt, theta=0.5, makeplot=True):
    ...

def define_command_line_options():
    ...

def read_command_line(use_argparse=True):
    if use_argparse:
        parser = define_command_line_options()
        args = parser.parse_args()
```



```
        print 'I={}, a={}, makeplot={}, dt_values={}'.format(
            args.I, args.a, args.makeplot, args.dt_values)
        return args.I, args.a, args.makeplot, args.dt_values
    else:
        if len(sys.argv) < 6:
            print 'Usage: %s I a on/off dt1 dt2 dt3 ...' % \
                  sys.argv[0]; sys.exit(1)

        I = float(sys.argv[1])
        a = float(sys.argv[2])
        T = float(sys.argv[3])
        makeplot = sys.argv[4] in ('on', 'True')
        dt_values = [float(arg) for arg in sys.argv[5:]]

        return I, a, makeplot, dt_values

def main():
    ...
```

This `decay_mod.py` file is already a module such that we can import functions in other programs. For example, we can in a file do

```
from decay_mod import solver
u, t = solver(I=1.0, a=3.0, T=3, dt=0.01, theta=0.5)
```

However, it should also be possible to both use `decay_mod.py` as a *and* execute the file as a program that runs `main()`. This is accomplished ending the file with a *test block*:

```
if __name__ == '__main__':
    main()
```

When `decay_mod.py` is used as a module, `__name__` equals the module `decay_mod`, while `__name__` equals `'__main__'` when the file is run as a gram. Optionally, we could run the verification tests if the word `verify` present on the command line and `verify_convergence_rate` could be called if `verify_rates` is found on the command line. The `verify_rates` argument be removed before we read parameter values from the command line, otherwise the `read_command_line` function (called by `main`) will not work properly

```
if __name__ == '__main__':
    if 'verify' in sys.argv:
        if verify_three_steps() and verify_discrete_solution():
            pass # ok
        else:
            print 'Bug in the implementation!'
    elif 'verify_rates' in sys.argv:
        sys.argv.remove('verify_rates')
        if not '--dt' in sys.argv:
            print 'Must assign several dt values'
            sys.exit(1)  # abort
        if verify_convergence_rate():
            pass
        else:
            print 'Bug in the implementation!'
```

```
    else:
        # Perform simulations
        main()
```

.2 Prefixing imported functions by the module name

nport statements of the form `from module import *` import functions and ariables in `module.py` into the current file. For example, when doing

```
rom numpy import *
rom matplotlib.pyplot import *
```

e get mathematical functions like `sin` and `exp` as well as MATLAB-style nctions like `linspace` and `plot`, which can be called by these well-known names. nfortunately, it sometimes becomes confusing to know where a particular nction comes from. Is it from `numpy`? Or `matplotlib.pyplot`? Or is it our wn function?

An alternative import is

```
mport numpy
mport matplotlib.pyplot
```

nd such imports require functions to be prefixed by the module name, e.g.,

```
 = numpy.linspace(0, T, Nt+1)
_e = I*numpy.exp(-a*t)
atplotlib.pyplot.plot(t, u_e)
```

his is normally regarded as a better habit because it is explicitly stated from hich module a function comes from.

The modules `numpy` and `matplotlib.pyplot` are so frequently used, and eir full names quite tedious to write, so two standard abbreviations have volved in the Python scientific computing community:

```
mport numpy as np
mport matplotlib.pyplot as plt

 = np.linspace(0, T, Nt+1)
_e = I*np.exp(-a*t)
lt.plot(t, u_e)
```

version of the `decay_mod` module where we use the `np` and `plt` prefixes is ound in the file `decay_mod_prefix.py`[13].

The downside of prefixing functions by the module name is that mathematical xpressions like $e^{-at}\sin(2\pi t)$ get cluttered with module names,

[13] http://tinyurl.com/jvzzcfn/softeng1/decay_mod_prefix.py


```
numpy.exp(-a*t)*numpy.sin(2(numpy.pi*t)
# or
np.exp(-a*t)*np.sin(2*np.pi*t)
```

Such an expression looks like `exp(-a*t)*sin(2*pi*t)` in most other p ming languages. Similarly, `np.linspace` and `plt.plot` look less fam people who are used to MATLAB and who have not adopted Python style. Whether to do `from module import *` or `import module` dep personal taste and the problem at hand. In these writings we use `from import` in shorter programs where similarity with MATLAB could be ad tage, and where a one-to-one correspondence between mathematical f and Python expressions is important. The style `import module` is p inside Python modules (see Exercise 5 for a demonstration).

4.3 Doctests

We have emphasized how important it is to be able to run tests in the pro any time. This was solved by calling various `verify*` functions in the p examples. However, there exists well-established procedures and corresp tools for automating the execution of tests. We shall briefly demonstr important techniques: *doctest* and *unit testing*. The corresponding files modules `decay_mod_doctest.py`[14] and `decay_mod_nosetest.py`[15].

A doc string (the first string after the function header) is used to do the purpose of functions and their arguments. Very often it is instru include an example on how to use the function. Interactive example Python shell are most illustrative as we can see the output resulting from calls. For example, we can in the `solver` function include an example o this function and printing the computed `u` and `t` arrays:

```
def solver(I, a, T, dt, theta):
    """
    Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt.


    >>> u, t = solver(I=0.8, a=1.2, T=4, dt=0.5, theta=0.5)
    >>> for t_n, u_n in zip(t, u):
    ...     print 't=%.1f, u=%.14f' % (t_n, u_n)
    t=0.0, u=0.80000000000000
    t=0.5, u=0.43076923076923
    t=1.0, u=0.23195266272189
    t=1.5, u=0.12489758761948
    t=2.0, u=0.06725254717972
    t=2.5, u=0.03621291001985
    t=3.0, u=0.01949925924146
    t=3.5, u=0.01049960113002
    t=4.0, u=0.00565363137770
    """
    ...
```

[14] http://tinyurl.com/jvzzcfn/softeng1/decay_mod_doctest.py
[15] http://tinyurl.com/jvzzcfn/softeng1/decay_mod_nosetest.py

When such interactive demonstrations are inserted in doc strings, Python's `doctest`[16] module can be used to automate running all commands in interactive essions and compare new output with the output appearing in the doc string. ll we have to do in the current example is to write

```
Terminal> python -m doctest decay_mod_doctest.py
```

his command imports the `doctest` module, which runs all tests. No additional ommand-line argument is allowed when running doctests. If any test fails, the roblem is reported, e.g.,

```
erminal> python -m doctest decay_mod_doctest.py
**********************************************************
ile "decay_mod_doctest.py", line 12, in decay_mod_doctest....
ailed example:
   for t_n, u_n in zip(t, u):
       print 't=%.1f, u=%.14f' % (t_n, u_n)
xpected:
   t=0.0, u=0.80000000000000
   t=0.5, u=0.43076923076923
   t=1.0, u=0.23195266272189
   t=1.5, u=0.12489758761948
   t=2.0, u=0.06725254717972
ot:
   t=0.0, u=0.80000000000000
   t=0.5, u=0.43076923076923
   t=1.0, u=0.23195266272189
   t=1.5, u=0.12489758761948
   t=2.0, u=0.06725254718756
**********************************************************
 items had failures:
  1 of   2 in decay_mod_doctest.solver
**Test Failed*** 1 failures.
```

Note that in the output of `t` and `u` we write `u` with 14 digits. Writing all 16 igits is not a good idea: if the tests are run on different hardware, round-off rors might be different, and the `doctest` module detects that the numbers are ot precisely the same and reports failures. In the present application, where $< u(t) \leq 0.8$, we expect round-off errors to be of size 10^{-16}, so comparing 15 igits would probably be reliable, but we compare 14 to be on the safe side.

Doctests are highly encouraged as they do two things: 1) demonstrate how a nction is used and 2) test that the function works.

Here is an example on a doctest in the `explore` function:

```
def explore(I, a, T, dt, theta=0.5, makeplot=True):
    """
    Run a case with the solver, compute error measure,
    and plot the numerical and exact solutions (if makeplot=True).

    >>> for theta in 0, 0.5, 1:
    ...     E = explore(I=1.9, a=2.1, T=5, dt=0.1, theta=theta,
    ...                 makeplot=False)
    ...     print '%.10E' % E
    ...
    7.3565079236E-02
    2.4183893110E-03
    6.5013039886E-02
    """
    ...
```

This time we limit the output to 10 digits.

Caution.

Doctests requires careful coding if they use command-line input or results to the terminal window. Command-line input must be simulat filling `sys.argv` correctly, e.g., `sys.argv = '-I 1.0 -a 5'.split`. output lines of print statements must be copied exactly as they ap when running the statements in an interactive Python shell.

4.4 Unit testing with nose

The unit testing technique consists of identifying small units of code, functions (or classes), and write one or more tests for each unit. One test ideally, not depend on the outcome of other tests. For example, the in function `solver` is a unit test, and the doctest in function `explore` but the latter depends on a working `solver`. Putting the error comp and plotting in `explore` in two separate functions would allow independ tests. In this way, the design of unit tests impacts the design of functio recommended practice is actually to design and write the unit tests f *then* implement the functions!

In scientific computing it is not always obvious how to best perfo testing. The units is naturally larger than in non-scientific software. Ve the solution procedure of a mathematical problem identifies a unit.

Basic use of nose. The `nose` package is a versatile tool for impler unit tests in Python. Here is a short explanation of the usage of nose:

1. Implement tests in functions with names starting with `test_`. Su tions cannot have any arguments.
2. The test functions perform assertions on computed results using functions from the `nose.tools` module.
3. The test functions can be in the source code files or be collected in s files with names `test*.py`.

[16] `http://docs.python.org/library/doctest.html`

ere comes a very simple illustration of the three points. Assume that we have is function in a module `mymod`:

```
def double(n):
    return 2*n
```

Either in this file, or in a separate file `test_mymod.py`, we implement a test nction whose purpose is to test that the function `double` works as intended:

```
import nose.tools as nt

def test_double():
    result = double(4)
    nt.assert_equal(result, 8)
```

otice that `test_double` has no arguments. We need to do an `import mymod` r `from mymod import double` if this test resides in a separate file. Running

```
erminal> nosetests -s mymod
```

akes the `nose` tool run all functions with names matching `test_*()` in `ymod.py`. Alternatively, if the test functions are in some `test_mymod.py` file, e can just write `nosetests -s`. The nose tool will then look for all files with ames mathching `test*.py` and run all functions `test_*()` in these files.

When you have nose tests in separate test files with names `test*.py` it is ommon to collect these files in a subdirectory `tests`, or `*_tests` if you have everal test subdirectories. Running `nosetests -s` will then recursively look for ll `tests` and `*_tests` subdirectories and run all functions `test_*()` in all files `est_*.py` in these directories. Just one command can then launch a series of sts in a directory tree!

An example of a `tests` directory with different types of `test*.py` files are ound in src/decay/tests[17]. Note that these perform imports of modules in the arent directory. These imports works well because the tests are supposed to be n by `nosetests -s` executed in the parent directory (`decay`).

Tip.

The `-s` option to `nosetests` assures that any print statement in the `test_*` functions appears in the output. Without this option, `nosetests` suppressed whatever the tests writes to the terminal window (standard output). Such behavior is annoying, especially when developing and testing tests.

[17] http://tinyurl.com/jvzzcfn/softeng1/tests

The number of failed tests and their details are reported, or an `OK` is if all tests passed.

The advantage with the `nose` package is two-fold:

1. tests are written and collected in a structured way, and
2. large collections of tests, scattered throughout a tree of directories executed with one command `nosetests -s`.

Alternative assert statements. In case the `nt.assert_equal` functi that the two arguments are equal, the test is a success, otherwise it is a and an exception of type `AssertionError` is raised. The particular exce the indicator that a test has failed.

Instead of calling the convenience function `nt.assert_equal`, we Python's plain `assert` statement, which tests if a boolean expression is t raises an `AssertionError` otherwise. Here, the statement is `assert res 8`.

A completely manual alternative is to explicitly raise an `Assertio` exception if the computed result is wrong:

```
if result != 8:
    raise AssertionError()
```

Applying nose. Let us illustrate how to use the `nose` tool for test functions in the `decay_mod` module. Or more precisely, the module i `decay_mod_unittest` with all the `verify*` functions removed as these outdated by the unit tests.

We design three unit tests:

1. A comparison between the computed u^n values and the exact solution.
2. A comparison between the computed u^n values and precomputed, reference values.
3. A comparison between observed and expected convergence rates.

These tests follow very closely the code in the previously shown `verify*` fu We start with comparing u^n, as computed by the function `solver`, to the for the exact discrete solution:

```
import nose.tools as nt
import decay_mod_unittest as decay_mod
import numpy as np

def exact_discrete_solution(n, I, a, theta, dt):
    """Return exact discrete solution of the theta scheme."""
```

```
    dt = float(dt)  # avoid integer division
    factor = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)
    return I*factor**n

def test_exact_discrete_solution():
    """
    Compare result from solver against
    formula for the discrete solution.
    """
    theta = 0.8; a = 2; I = 0.1; dt = 0.8
    N = int(8/dt)  # no of steps
    u, t = decay_mod.solver(I=I, a=a, T=N*dt, dt=dt, theta=theta)
    u_de = np.array([exact_discrete_solution(n, I, a, theta, dt)
                     for n in range(N+1)])
    diff = np.abs(u_de - u).max()
    nt.assert_almost_equal(diff, 0, delta=1E-14)
```

The `nt.assert_almost_equal` is the relevant function for comparing two eal numbers. The `delta` argument specifies a tolerance for the comparison. lternatively, one can specify a `places` argument for the number of decimal laces to be used in the comparison.

After having carefully verified the implementation, we may store correctly omputed numbers in the test program or in files for use in future tests. Here is n example on how the outcome from the `solver` function can be compared to hat is considered to be correct results:

```
def test_solver():
    """
    Compare result from solver against
    precomputed arrays for theta=0, 0.5, 1.
    """
    I=0.8; a=1.2; T=4; dt=0.5  # fixed parameters
    precomputed = {
        't': np.array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,
                        3. ,  3.5,  4. ]),
        0.5: np.array(
            [ 0.8       ,  0.43076923,  0.23195266, 0.12489759,
              0.06725255,  0.03621291,  0.01949926, 0.0104996 ,
              0.00565363]),
        0: np.array(
            [  8.00000000e-01,   3.20000000e-01,
               1.28000000e-01,   5.12000000e-02,
               2.04800000e-02,   8.19200000e-03,
               3.27680000e-03,   1.31072000e-03,
               5.24288000e-04]),
        1: np.array(
            [ 0.8       ,  0.5       ,  0.3125    ,  0.1953125 ,
              0.12207031,  0.07629395,  0.04768372,  0.02980232,
              0.01862645]),
        }
    for theta in 0, 0.5, 1:
        u, t = decay_mod.solver(I, a, T, dt, theta=theta)
        diff = np.abs(u - precomputed[theta]).max()
        # Precomputed numbers are known to 8 decimal places
        nt.assert_almost_equal(diff, 0, places=8,
                               msg='theta=%s' % theta)
```


The `precomputed` object is a dictionary with four keys: `'t'` for the tim and three θ values for u^n solutions corresponding to $\theta = 0, 0.5, 1$.

Testing for special type of input data that may cause trouble con a common way of constructing unit tests. For example, the updating for u^{n+1} may be incorrectly evaluated because of unintended integer d With

```
theta = 1; a = 1; I = 1; dt = 2
```

the nominator and denominator in the updating expression,

```
(1 - (1-theta)*a*dt)
(1 + theta*dt*a)
```

evaluate to 1 and 3, respectively, and the fraction `1/3` will call up integer and consequently lead to `u[n+1]=0`. We construct a unit test to ma `solver` is smart enough to avoid this problem:

```
def test_potential_integer_division():
    """Choose variables that can trigger integer division."""
    theta = 1; a = 1; I = 1; dt = 2
    N = 4
    u, t = decay_mod.solver(I=I, a=a, T=N*dt, dt=dt, theta=theta)
    u_de = np.array([exact_discrete_solution(n, I, a, theta, dt)
                     for n in range(N+1)])
    diff = np.abs(u_de - u).max()
    nt.assert_almost_equal(diff, 0, delta=1E-14)
```

The final test is to see that the convergence rates corresponding to $\theta =$ are 1, 2, and 1, respectively:

```
def test_convergence_rates():
    """Compare empirical convergence rates to exact ones."""
    # Set command-line arguments directly in sys.argv
    import sys
    sys.argv[1:] = '--I 0.8 --a 2.1 --T 5 '\
                   '--dt 0.4 0.2 0.1 0.05 0.025'.split()
    r = decay_mod.main()
    for theta in r:
        nt.assert_true(r[theta])  # check for non-empty list

    expected_rates = {0: 1, 1: 1, 0.5: 2}
    for theta in r:
        r_final = r[theta][-1]
        # Compare to 1 decimal place
        nt.assert_almost_equal(expected_rates[theta], r_final,
                               places=1, msg='theta=%s' % theta)
```

Nothing more is needed in the `test_decay_nose.py`[18] file where t reside. Running `nosetests -s` will report `Ran 3 tests` and an `OK` for Every time we modify the `decay_mod_unittest` module we can run `nos` to quickly see if the edits have any impact on the verification tests.

[18] `http://tinyurl.com/jvzzcfn/softeng1/tests/test_decay_nose.py`

nstallation of nose. The `nose` package does not come with a standard ython distribution and must therefore be installed separately. The procedure is tandard and described on Nose's web pages[19]. On Debian-based Linux systems ne command is `sudo apt-get install python-nose`, and with MacPorts you un `sudo port install py27-nose`.

Using nose to test modules with doctests. Assume that `mod` is the name f some module that contains doctests. We may let `nose` run these doctests and eport errors in the standard way using the code set-up

```
import doctest
import mod

def test_mod():
    failure_count, test_count = doctest.testmod(m=mod)
    nt.assert_equal(failure_count, 0,
                    msg='%d tests out of %d failed' %
                    (failure_count, test_count))
```

he call to `doctest.testmod` runs all doctests in the module file `mod.py` and eturns the number of failures (`failure_count`) and the total number of tests `est_count`). A real example is found in the file `test_decay_doctest.py`[20].

.5 Classical class-based unit testing

he classical way of implementing unit tests derives from the JUnit tool in Java here all tests are methods in a class for testing. Python comes with a module `nittest` for doing this type of unit tests. While `nose` allows simple functions for nit tests, `unittest` requires deriving a class `Test*` from `unittest.TestCase` nd implementing each test as methods with names `test_*` in that class. I trongly recommend to use `nose` over `unittest`, because it is much simpler and ore convenient, but class-based unit testing is a very classical subject that omputational scientists should have some knowledge about. That is why a short ntroduction to `unittest` is included below.

Basic use of unittest. We apply the `double` function in the `mymod` module ntroduced in the previous section as example. Unit testing with the aid of the `nittest` module consists of writing a file `test_mymod.py` with the content

```
import unittest
import mymod

class TestMyCode(unittest.TestCase):
    def test_double(self):
        result = mymod.double(4)
        self.assertEqual(result, 8)
```

[19] http://nose.readthedocs.org/en/latest/
[20] http://tinyurl.com/jvzzcfn/softeng1/tests/test_decay_doctest.py


```
if __name__ == '__main__':
    unittest.main()
```

The test is run by executing the test file `test_mymod.py` as a standard program. There is no support in `unittest` for automatically locat running all tests in all test files in a directory tree.

Those who have experience with object-oriented programming will the difference between using `unittest` and `nose` is minor.

Demonstration of unittest. The same tests as shown for the nose fra are reimplemented with the `TestCase` classes in the file `test_decay_un` `py`[21]. The tests are identical, the only difference being that with `unit` must write the tests as methods in a class and the assert functions have different names.

```
import unittest
import decay_mod_unittest as decay
import numpy as np

def exact_discrete_solution(n, I, a, theta, dt):
    factor = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)
    return I*factor**n

class TestDecay(unittest.TestCase):

    def test_exact_discrete_solution(self):
        ...
        diff = np.abs(u_de - u).max()
        self.assertAlmostEqual(diff, 0, delta=1E-14)

    def test_solver(self):
        ...
        for theta in 0, 0.5, 1:
            ...
            self.assertAlmostEqual(diff, 0, places=8,
                                   msg='theta=%s' % theta)

    def test_potential_integer_division():
        ...
        self.assertAlmostEqual(diff, 0, delta=1E-14)

    def test_convergence_rates(self):
        ...
        for theta in r:
            ...
            self.assertAlmostEqual(...)

if __name__ == '__main__':
    unittest.main()
```

[21] http://tinyurl.com/jvzzcfn/softeng1/tests/test_decay_nose.py

.6 Implementing simple problem and solver classes

he θ-rule was compactly and conveniently implemented in a function `solver` ı Section **??**. In more complicated problems it might be beneficial to use lasses and introduce a class `Problem` to hold the definition of the physical roblem, a class `Solver` to hold the data and methods needed to numerically olve the problem, and a class `Visualizer` to make plots. This idea will now be lustrated, resulting in code that represents an alternative to the `solver` and `xplore` functions found in the `decay_mod` module.

Explaining the details of class programming in Python is considered beyond ıe scope of this text. Readers who are unfamiliar with Python class programming ıould first consult one of the many electronic Python tutorials or textbooks ɔ come up to speed with concepts and syntax of Python classes before reading n. The author has a gentle introduction to class programming for scientific pplications in [1], see Chapter 7 and 9 and Appendix E. Other useful resources re

- The Python Tutorial: `http://docs.python.org/2/tutorial/classes.html`
- Wiki book on Python Programming: `http://en.wikibooks.org/wiki/Python_Programming/Classes`
- tutorialspoint.com: `http://www.tutorialspoint.com/python/python_classes_objects.htm`

'he problem class. The purpose of the problem class is to store all infor- ıation about the mathematical model. This usually means all the physical arameters in the problem. In the current example with exponential decay we ıay also add the exact solution of the ODE to the problem class. The simplest ɔrm of a problem class is therefore

```
rom numpy import exp

lass Problem:
    def __init__(self, I=1, a=1, T=10):
        self.T, self.I, self.a = I, float(a), T

    def u_exact(self, t):
        I, a = self.I, self.a
        return I*exp(-a*t)
```

Ve could in the `u_exact` method have written `self.I*exp(-self.a*t)`, but sing local variables `I` and `a` allows the formula `I*exp(-a*t)` which looks closer ɔ the mathematical expression Ie^{-at}. This is not an important issue with the ırrent compact formula, but is beneficial in more complicated problems with ɔnger formulas to obtain the closest possible relationship between code and ıathematics. My coding style is to strip off the `self` prefix when the code xpresses mathematical formulas.

The class data can be set either as arguments in the constructor o time later, e.g.,

```
problem = Problem(T=5)
problem.T = 8
problem.dt = 1.5
```

(Some programmers prefer `set` and `get` functions for setting and getting classes, often implemented via *properties* in Python, but I consider that when we just have a few data items in a class.)

It would be convenient if class `Problem` could also initialize the data f command line. To this end, we add a method for defining a set of comm options and a method that sets the local attributes equal to what was f the command line. The default values associated with the command-line are taken as the values provided to the constructor. Class `Problem` now

```
class Problem:
    def __init__(self, I=1, a=1, T=10):
        self.T, self.I, self.a = I, float(a), T

    def define_command_line_options(self, parser=None):
        if parser is None:
            import argparse
            parser = argparse.ArgumentParser()

        parser.add_argument(
            '--I', '--initial_condition', type=float,
            default=self.I, help='initial condition, u(0)',
            metavar='I')
        parser.add_argument(
            '--a', type=float, default=self.a,
            help='coefficient in ODE', metavar='a')
        parser.add_argument(
            '--T', '--stop_time', type=float, default=self.T,
            help='end time of simulation', metavar='T')
        return parser

    def init_from_command_line(self, args):
        self.I, self.a, self.T = args.I, args.a, args.T

    def exact_solution(self, t):
        I, a = self.I, self.a
        return I*exp(-a*t)
```

Observe that if the user already has an `ArgumentParser` object it can be s but if she does not have any, class `Problem` makes one. Python's `None` used to indicate that a variable is not initialized with a proper value.

The solver class. The solver class stores data related to the numerical method and provides a function `solve` for solving the problem. A proble must be given to the constructor so that the solver can easily look up data. In the present example, the data related to the numerical solution consists of Δt and θ. We add, as in the problem class, functionality for Δt and θ from the command line:

```
class Solver:
    def __init__(self, problem, dt=0.1, theta=0.5):
        self.problem = problem
        self.dt, self.theta = float(dt), theta

    def define_command_line_options(self, parser):
        parser.add_argument(
            '--dt', '--time_step_value', type=float,
            default=0.5, help='time step value', metavar='dt')
        parser.add_argument(
            '--theta', type=float, default=0.5,
            help='time discretization parameter', metavar='dt')
        return parser

    def init_from_command_line(self, args):
        self.dt, self.theta = args.dt, args.theta

    def solve(self):
        from decay_mod import solver
        self.u, self.t = solver(
            self.problem.I, self.problem.a, self.problem.T,
            self.dt, self.theta)

    def error(self):
        u_e = self.problem.exact_solution(self.t)
        e = u_e - self.u
        E = sqrt(self.dt*sum(e**2))
        return E
```

ote that we here simply reuse the implementation of the numerical method from ıe `decay_mod` module. The `solve` function is just a *wrapper* of the previously eveloped stand-alone `solver` function.

'he visualizer class. The purpose of the visualizer class is to plot the numer-al solution stored in class `Solver`. We also add the possibility to plot the exact olution. Access to the problem and solver objects is required when making plots the constructor must hold references to these objects:

```
class Visualizer:
    def __init__(self, problem, solver):
        self.problem, self.solver = problem, solver

    def plot(self, include_exact=True, plt=None):
        """
        Add solver.u curve to the plotting object plt,
        and include the exact solution if include_exact is True.
        This plot function can be called several times (if
        the solver object has computed new solutions).
        """
        if plt is None:
            import scitools.std  as plt # can use matplotlib as well

        plt.plot(self.solver.t, self.solver.u, '--o')
        plt.hold('on')
        theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
        name = theta2name.get(self.solver.theta, '')
        legends = ['numerical %s' % name]
```



```
        if include_exact:
            t_e = linspace(0, self.problem.T, 1001)
            u_e = self.problem.exact_solution(t_e)
            plt.plot(t_e, u_e, 'b-')
            legends.append('exact')
        plt.legend(legends)
        plt.xlabel('t')
        plt.ylabel('u')
        plt.title('theta=%g, dt=%g' %
                  (self.solver.theta, self.solver.dt))
        plt.savefig('%s_%g.png' % (name, self.solver.dt))
        return plt
```

The `plt` object in the `plot` method is worth a comment. The idea `plot` can add a numerical solution curve to an existing plot. Calling `pl` a `plt` object (which has to be a `matplotlib.pyplot` or `scitools.st` in this implementation), will just add the curve `self.solver.u` as a line with circles at the mesh points (leaving the color of the curve u plotting tool). This functionality allows plots with several solutions: ju a loop where new data is set in the problem and/or solver classes, the `solve()` method is called, and the most recent numerical solution is pl the `plot(plt)` method in the visualizer object Exercise 6 describes a setting where this functionality is explored.

Combining the objects. Eventually we need to show how the classes `Solver`, and `Visualizer` play together:

```
def main():
    problem = Problem()
    solver = Solver(problem)
    viz = Visualizer(problem, solver)

    # Read input from the command line
    parser = problem.define_command_line_options()
    parser = solver. define_command_line_options(parser)
    args = parser.parse_args()
    problem.init_from_command_line(args)
    solver. init_from_command_line(args)

    # Solve and plot
    solver.solve()
    import matplotlib.pyplot as plt
    #import scitools.std as plt
    plt = viz.plot(plt=plt)
    E = solver.error()
    if E is not None:
        print 'Error: %.4E' % E
    plt.show()
```

The file `decay_class.py`[22] constitutes a module with the three clas the `main` function.

[22] http://tinyurl.com/jvzzcfn/softeng1/decay_class.py

Test the understanding.

Implement the problem in Exercise **??** in terms of problem, solver, and visualizer classes. Equip the classes and their methods with doc strings with tests. Also include nose tests.

.7 Improving the problem and solver classes

he previous `Problem` and `Solver` classes containing parameters soon get much epetitive code when the number of parameters increases. Much of this code can e parameterized and be made more compact. For this purpose, we decide to ollect all parameters in a dictionary, `self.prms`, with two associated dictionaries `elf.types` and `self.help` for holding associated object types and help strings. rovided a problem, solver, or visualizer class defines these three dictionaries in ıe constructor, using default or user-supplied values of the parameters, we can eate a super class `Parameters` with general code for defining command-line ptions and reading them as well as methods for setting and getting a parameter. `Problem` or `Solver` class will then inherit command-line functionality and the et/get methods from the `Parameters` class.

generic class for parameters. A simplified version of the parameter class ooks as follows:

```
class Parameters:
    def set(self, **parameters):
        for name in parameters:
            self.prms[name] = parameters[name]

    def get(self, name):
        return self.prms[name]

    def define_command_line_options(self, parser=None):
        if parser is None:
            import argparse
            parser = argparse.ArgumentParser()

        for name in self.prms:
            tp = self.types[name] if name in self.types else str
            help = self.help[name] if name in self.help else None
            parser.add_argument(
                '--' + name, default=self.get(name), metavar=name,
                type=tp, help=help)

        return parser

    def init_from_command_line(self, args):
        for name in self.prms:
            self.prms[name] = getattr(args, name)
```


The file `class_decay_oo.py`[23] contains a slightly more advanced version `Parameters` where we in the `set` and `get` functions test for valid paramete and raise exceptions with informative messages if any name is not regis

The problem class. A class `Problem` for the problem $u' = -au$, u $t \in (0, T]$, with parameters input a, I, and T can now be coded as

```
class Problem(Parameters):
    """
    Physical parameters for the problem u'=-a*u, u(0)=I,
    with t in [0,T].
    """
    def __init__(self):
        self.prms = dict(I=1, a=1, T=10)
        self.types = dict(I=float, a=float, T=float)
        self.help = dict(I='initial condition, u(0)',
                         a='coefficient in ODE',
                         T='end time of simulation')

    def exact_solution(self, t):
        I, a = self.get('I'), self.get('a')
        return I*np.exp(-a*t)
```

The solver class. Also the solver class is derived from class `Para` and works with the `prms`, `types`, and `help` dictionaries in the same class `Problem`. Otherwise, the code is very similar to class `Solver` `decay_class.py` file:

```
class Solver(Parameters):
    def __init__(self, problem):
        self.problem = problem
        self.prms = dict(dt=0.5, theta=0.5)
        self.types = dict(dt=float, theta=float)
        self.help = dict(dt='time step value',
                         theta='time discretization parameter')

    def solve(self):
        from decay_mod import solver
        self.u, self.t = solver(
            self.problem.get('I'),
            self.problem.get('a'),
            self.problem.get('T'),
            self.get('dt'),
            self.get('theta'))

    def error(self):
        try:
            u_e = self.problem.exact_solution(self.t)
            e = u_e - self.u
            E = np.sqrt(self.get('dt')*np.sum(e**2))
        except AttributeError:
            E = None
        return E
```

[23] `http://tinyurl.com/jvzzcfn/softeng1/class_decay_oo.py`

'he visualizer class. Class `Visualizer` can be identical to the one in the `ecay_class.py` file since the class does not need any parameters. However, a :w adjustments in the `plot` method is necessary since parameters are accessed s, e.g., `problem.get('T')` rather than `problem.T`. The details are found in ıe file `class_decay_oo.py`.

Finally, we need a function that solves a real problem using the classes `roblem`, `Solver`, and `Visualizer`. This function can be just like `main` in the `ecay_class.py` file.

The advantage with the `Parameters` class is that it scales to problems with large number of physical and numerical parameters: as long as the parameters re defined once via a dictionary, the compact code in class `Parameters` can andle any collection of parameters of any size.

Performing scientific experiments

Goal.

This section explores the behavior of a numerical method for a differential equation through computer experiments. In particular, it is shown how scientific experiments can be set up and reported. We address the ODE problem

$$u'(t) = -au(t), \quad u(0) = I, \quad t \in (0, T], \tag{5}$$

numerically discretized by the θ-rule:

$$u^{n+1} = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}u^n, \quad u^0 = I\,.$$

Our aim is to plot $u^0, u^1, \ldots, u^N$ together with the exact solution $u_{\mathrm{e}} = Ie^{-at}$ for various choices of the parameters in this numerical problem: I, a, Δt, and θ. We are especially interested in how the discrete solution compares with the exact solution when the Δt parameter is varied and θ takes on the three values corresponding to the Forward Euler, Backward Euler, and Crank-Nicolson schemes ($\theta = 0, 1, 0.5$, respectively).

.1 Software

verified implementation for computing the numerical solution u^n and plotting together with the exact solution u_{e} is found in the file `decay_mod.py`[24]. This rogram admits command-line arguments to specify a series of Δt values and ill run a loop over these values and $\theta = 0, 0.5, 1$. We make a slight edit of how ıe plots are designed: the numerical solution is specified with line type `'r-o'` lashed red lines with dots at the mesh points), and the `show()` command is

[24] `http://tinyurl.com/jvzzcfn/softeng1/decay_mod.py`

removed to avoid a lot of plot windows popping up on the computer scr hardcopies of the plot are still stored in files via `savefig`). The slightly r program has the name `experiments/decay_mod.py`[25]. All files associat the scientific investigation are collected in a subdirectory `experiments`

Running the experiments is easy since the `decay_mod.py` program has the loops over θ and Δt implemented. An experiment with $I = 1$ $T = 5$, and $dt = 0.5, 0.25, 0.1, 0.05$ is run by

```
Terminal> python decay_mod.py --I 1 --a 2 --makeplot \
          --T 5 --dt 0.5 0.25 0.1 0.05
```

5.2 Combining plot files

The `decay_mod.py` program generates a lot of image files, e.g., `FE` `BE_*.png`, and `CN_*.png`. We want to combine all the `FE_*.png` fi table fashion in one file, with two images in each row, starting with the Δt in the upper left corner and decreasing the value as we go to the ri down. This can be done using the `montage`[26] program. The often occurri areas around the plots can be cropped away by the `convert -trim` co The remaining white can be made transparent for HTML pages with a nc background by the command `convert -transparent white`.

Also plot files in the PDF format with names `FE_*.pdf`, `BE_*.p` `CN_*.pdf` are generated and these should be combined using other tools to combine individual plots into one file with one plot per page, and to combine the pages into a table with multiple plots per page. The r image often has some extra surrounding white space that can be remove `pdfcrop` program. The code snippets below contain all details about th of `montage`, `convert`, `pdftk`, `pdfnup`, and `pdfcrop`.

Running manual commands is boring, and errors may easily sneak i for automating manual work and documenting the operating system co we actually issued in the experiment, we should write a *script* (little pr An alternative is to write the commands into an IPython notebook the notebook as the script. A plain script as a standard Python progr separate text file will be used here.

Reproducible science.

A script that automates running our computer experiments will ensure the experiments can easily be rerun by ourselves or others in the fu either to check the results or redo the experiments with other input Also, whatever we did to produce the results is documented in every

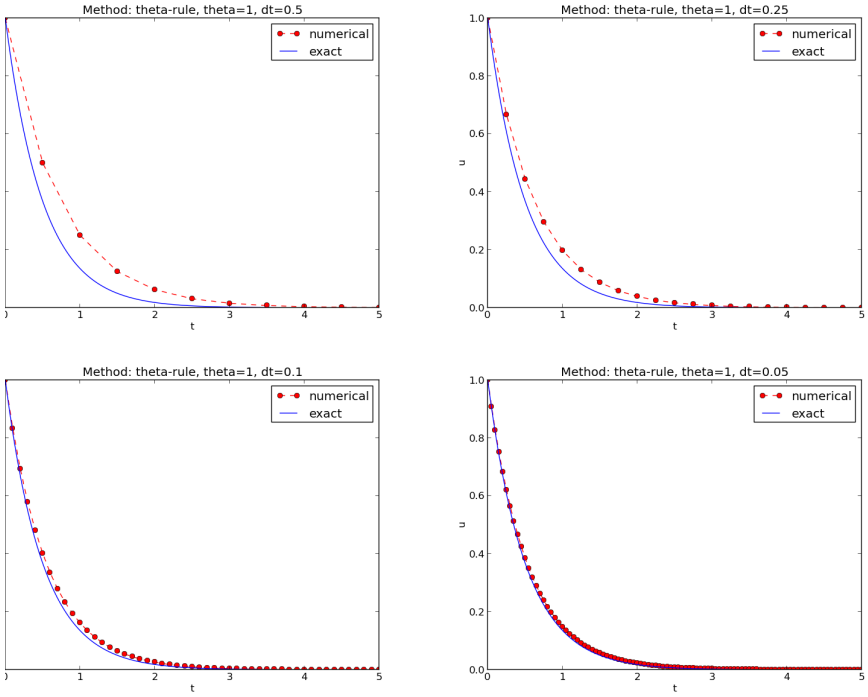
[25] `http://tinyurl.com/jvzzcfn/softeng1/experiments/decay_mod.py`
[26] `http://www.imagemagick.org/script/montage.php`

in the script. Automating scripts are therefore essential to making our research *reproducible*, which is a fundamental principle in science.

The script takes a list of Δt values on the command line as input and akes three combined images, one for each θ value, displaying the quality of the umerical solution as Δt varies. For example,

```
erminal> python decay_exper0.py 0.5 0.25 0.1 0.05
```

esults in images `FE.png`, `CN.png`, `BE.png`, `FE.pdf`, `CN.pdf`, and `BE.pdf`, each ith four plots corresponding to the four Δt values. Each plot compares the umerical solution with the exact one. The latter image is shown in Figure 2.



igure 2: Illustration of the Backward Euler method for four time step values.

Ideally, the script should be scalable in the sense that it works for any number f Δt values, which is the case for this particular implementation:


```
import os, sys

def run_experiments(I=1, a=2, T=5):
    # The command line must contain dt values
    if len(sys.argv) > 1:
        dt_values = [float(arg) for arg in sys.argv[1:]]
    else:
        print 'Usage: %s dt1 dt2 dt3 ...' %  sys.argv[0]
        sys.exit(1)  # abort

    # Run module file as a stand-alone application
    cmd = 'python decay_mod.py --I %g --a %g --makeplot --T %g'
          (I, a, T)
    dt_values_str = ' '.join([str(v) for v in dt_values])
    cmd += ' --dt %s' % dt_values_str
    print cmd
    failure = os.system(cmd)
    if failure:
        print 'Command failed:', cmd; sys.exit(1)

    # Combine images into rows with 2 plots in each row
    image_commands = []
    for method in 'BE', 'CN', 'FE':
        pdf_files = ' '.join(['%s_%g.pdf' % (method, dt)
                              for dt in dt_values])
        png_files = ' '.join(['%s_%g.png' % (method, dt)
                              for dt in dt_values])
        image_commands.append(
            'montage -background white -geometry 100%' +
            ' -tile 2x %s %s.png' % (png_files, method))
        image_commands.append(
            'convert -trim %s.png %s.png' % (method, method))
        image_commands.append(
            'convert %s.png -transparent white %s.png' %
            (method, method))
        image_commands.append(
            'pdftk %s output tmp.pdf' % pdf_files)
        num_rows = int(round(len(dt_values)/2.0))
        image_commands.append(
            'pdfnup --nup 2x%d tmp.pdf' % num_rows)
        image_commands.append(
            'pdfcrop tmp-nup.pdf %s.pdf' % method)

    for cmd in image_commands:
        print cmd
        failure = os.system(cmd)
        if failure:
            print 'Command failed:', cmd; sys.exit(1)

    # Remove the files generated above and by decay_mod.py
    from glob import glob
    filenames = glob('*_*.png') + glob('*_*.pdf') + \
                glob('*_*.eps') + glob('tmp*.pdf')
    for filename in filenames:
        os.remove(filename)

if __name__ == '__main__':
    run_experiments()
```

his file is available as `experiments/decay_exper0.py`[27].

We may comment upon many useful constructs in this script:

- `[float(arg) for arg in sys.argv[1:]]` builds a list of real numbers from all the command-line arguments.
- `failure = os.system(cmd)` runs an operating system command, e.g., another program. The execution is successful only if `failure` is zero.
- Unsuccessful execution usually makes it meaningless to continue the program, and therefore we abort the program with `sys.exit(1)`. Any argument different from 0 signifies to the computer's operating system that our program stopped with a failure.
- `['%s_%s.png' % (method, dt) for dt in dt_values]` builds a list of filenames from a list of numbers (`dt_values`).
- All `montage`, `convert`, `pdftk`, `pdfnup`, and `pdfcrop` commands for creating composite figures are stored in a list and later executed in a loop.
- `glob('*_*.png')` returns a list of the names of all files in the current directory where the filename matches the Unix wildcard notation[28] `*_*.png` (meaning any text, underscore, any text, and then `.png`).
- `os.remove(filename)` removes the file with name `filename`.

.3 Interpreting output from other programs

rograms that run other programs, like `decay_exper0.py` does, will often need o interpret output from those programs. Let us demonstrate how this is done ı Python by extracting the relations between θ, Δt, and the error E as written o the terminal window by the `decay_mod.py` program, when being executed by `ecay_exper0.py`. We will

- read the output from the `decay_mod.py` program
- interpret this output and store the E values in arrays for each θ value
- plot E versus Δt, for each θ, in a log-log plot

The simple `os.system(cmd)` call does not allow us to read the output from unning `cmd`. Instead we need to invoke a bit more involved procedure:

[27] `http://tinyurl.com/jvzzcfn/softeng1/experiments/decay_exper0.py`
[28] `http://en.wikipedia.org/wiki/Glob_(programming)`


```
from subprocess import Popen, PIPE, STDOUT
p = Popen(cmd, shell=True, stdout=PIPE, stderr=STDOUT)
output, dummy = p.communicate()
failure = p.returncode
if failure:
    print 'Command failed:', cmd; sys.exit(1)
```

The command stored in `cmd` is run and all text that is written to the s output *and* the standard error is available in the string `output`. Or i words, the text in `output` is what appeared in the terminal window while `cmd`.

Our next task is to run through the `output` string, line by line, an current line prints θ, Δt, and E, we split the line into these three pie store the data. The chosen storage structure is a dictionary `errors` w `dt` to hold the Δt values in a list, and three θ keys to hold the corresp values in a list. The relevant code lines are

```
errors = {'dt': dt_values, 1: [], 0: [], 0.5: []}
for line in output.splitlines():
    words = line.split()
    if words[0] in ('0.0', '0.5', '1.0'):  # line with E?
        # typical line: 0.0   1.25:    7.463E+00
        theta = float(words[0])
        E = float(words[2])
        errors[theta].append(E)
```

Note that we do not bother to store the Δt values as we read them from because we already have these values in the `dt_values` list.

We are now ready to plot E versus Δt for $\theta = 0, 0.5, 1$:

```
import matplotlib.pyplot as plt
plt.loglog(errors['dt'], errors[0], 'ro-')
plt.hold('on')
plt.loglog(errors['dt'], errors[0.5], 'b+-')
plt.loglog(errors['dt'], errors[1], 'gx-')
plt.legend(['FE', 'CN', 'BE'], loc='upper left')
plt.xlabel('log(time step)')
plt.ylabel('log(error)')
plt.title('Error vs time step')
plt.savefig('error.png')
plt.savefig('error.pdf')
```

Plots occasionally need some manual adjustments. Here, the axis of the plot look nicer if we adapt them strictly to the data, see Figure 3. To t we need to compute $\min E$ and $\max E$, and later specify the extent of t

```
# Find min/max for the axis
E_min = 1E+20; E_max = -E_min
for theta in 0, 0.5, 1:
    E_min = min(E_min, min(errors[theta]))
    E_max = max(E_max, max(errors[theta]))

plt.loglog(errors['dt'], errors[0], 'ro-')
```

```
...
lt.axis([min(dt_values), max(dt_values), E_min, E_max])
...
```

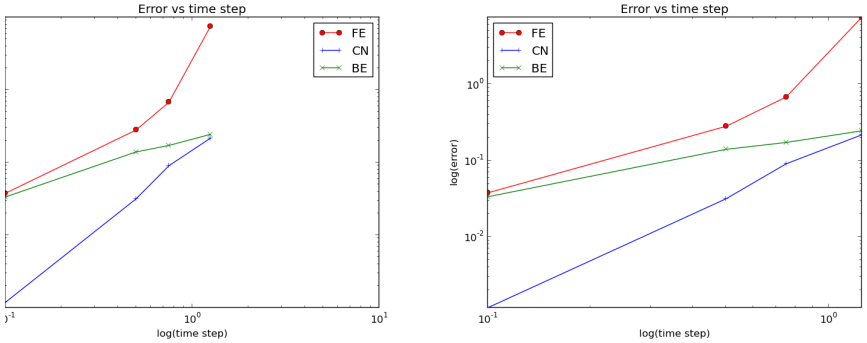


Figure 3: Default plot (left) and manually adjusted axes (right).

The complete program, incorporating the code snippets above, is found in `xperiments/decay_exper1.py`[29]. This example can hopefully act as template or numerous other occasions where one needs to run experiments, extract data om the output of programs, make plots, and combine several plots in a figure le. The `decay_exper1.py` program is organized as a module, and other files an then easily extend the functionality, as illustrated in the next section.

.4 Making a report

he results of running computer experiments are best documented in a little eport containing the problem to be solved, key code segments, and the plots om a series of experiments. At least the part of the report containing the lots should be automatically generated by the script that performs the set of xperiments, because in that script we know exactly which input data that were sed to generate a specific plot, thereby ensuring that each figure is connected to ne right data. Take a look at an example at `http://tinyurl.com/k3sdbuv/riting_reports//sphinx-cloud/` to see what we have in mind.

lain HTML. Scientific reports can be written in a variety of formats. Here e begin with the HTML[30] format which allows efficient viewing of all the xperiments in any web browser. The program `decay_exper1_html.py`[31] calls `ecay_exper1.py` to perform the experiments and then runs statements for cre- ting an HTML file with a summary, a section on the mathematical problem, a ection on the numerical method, a section on the `solver` function implementing

[29] `http://tinyurl.com/jvzzcfn/softeng1/experiments/decay_exper1.py`
[30] `http://en.wikipedia.org/wiki/HTML`
[31] `http://tinyurl.com/jvzzcfn/softeng1/experiments/decay_exper1_html.py`

the method, and a section with subsections containing figures that sl results of experiments where Δt is varied for $\theta = 0, 0.5, 1$. The mentioned file contains all the details for writing this HTML report[32]. You can viev port on `http://tinyurl.com/k3sdbuv/writing_reports//_static/r html.html`.

HTML with MathJax. Scientific reports usually need mathemat mulas and hence mathematical typesetting. In plain HTML, as usec `decay_exper1_html.py` file, we have to use just the keyboard chara write mathematics. However, there is an extension to HTML, calle Jax[33], which allows formulas and equations to be typeset with LaTeX and nicely rendered in web browsers, see Figure 4. A relatively smal of LaTeX environments is supported, but the syntax for formulas is qu Inline formulas are look like `\( u'=-au \)` while equations are surrounde signs. Inside such signs, one can use `\[ u'=-au \]` for unnumbered eq or `\begin{equation}` and `\end{equation}` surrounding `u'=-au` for nu equations, or `\begin{align}` and `\end{align}` for multiple aligned eq You need to be familiar with mathematical typesetting in LaTeX[34].

The file `decay_exper1_mathjax.py`[35] contains all the details for turi previous plain HTML report into web pages with nicely typeset math The corresponding HTML code[36] be studied to see all details of the mathe typesetting.

We address the initial-value problem

$$u'(t) = -au(t), \quad t \in (0,T],$$
$$u(0) = I,$$

where a, I, and T are prescribed parameters, and $u(t)$ is the unknown function to be estimated. This mathematical model is relevant for physical phenomena featuring exponential decay in time.

Numerical solution method

We introduce a mesh in time with points $0 = t_0 < t_1 \cdots < t_N = T$. For simplicity, we assume constant spacing Δt between the mesh points: $\Delta t = t_n - t_{n-1}$, $n = 1, \ldots, N$. Let u^n be the numerical approximation to the exact solution at t_n. The θ-rule is used to solve (1) numerically:

$$u^{n+1} = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}u^n,$$

for $n = 0, 1, \ldots, N-1$. This scheme corresponds to

- The Forward Euler scheme when $\theta = 0$
- The Backward Euler scheme when $\theta = 1$
- The Crank-Nicolson scheme when $\theta = 1/2$

Implementation

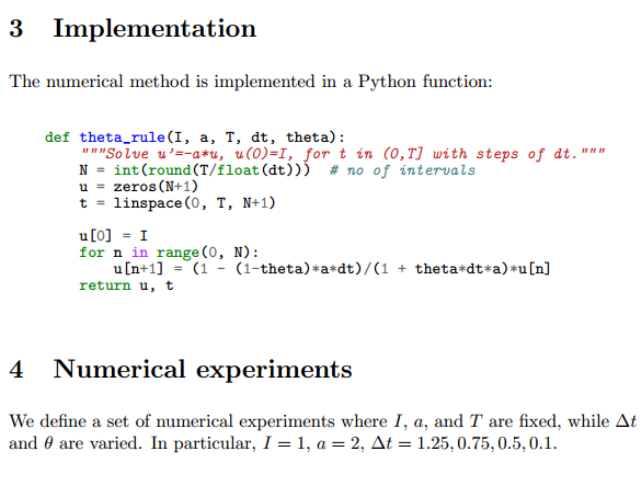
The numerical method is implemented in a Python function:

```
def theta_rule(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    N = int(round(T/float(dt)))  # no of intervals
    u = zeros(N+1)
    t = linspace(0, T, N+1)
```

Figure 4: Report in HTML format with MathJax.

[32] `http://tinyurl.com/k3sdbuv/writing_reports//_static/report_html.html.h`
[33] `http://www.mathjax.org/`
[34] `http://en.wikibooks.org/wiki/LaTeX/Mathematics`
[35] `http://tinyurl.com/jvzzcfn/softeng1/experiments/decay_exper1_html.py`
[36] `http://tinyurl.com/k3sdbuv/writing_reports//_static/report_mathjax.htm`

LaTeX. The *de facto* language for mathematical typesetting and scientific report writing is LaTeX[37]. A number of very sophisticated packages have been added to the language over a period of three decades, allowing very fine-tuned layout and typesetting. For output in the PDF format[38], see Figure 5 for an example, LaTeX is the definite choice when it comes to quality. The LaTeX language used to write the reports has typically a lot of commands involving backslashes and braces[39]. For output on the web, using HTML (and not the PDF directly in the browser window), LaTeX struggles with delivering high quality typesetting. Other tools, especially Sphinx, give better results and can also produce nice-looking PDFs. The file `decay_exper1_latex.py` shows how to generate the LaTeX source from a program.



3 Implementation

The numerical method is implemented in a Python function:

```
def theta_rule(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    N = int(round(T/float(dt)))  # no of intervals
    u = zeros(N+1)
    t = linspace(0, T, N+1)

    u[0] = I
    for n in range(0, N):
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t
```

4 Numerical experiments

We define a set of numerical experiments where I, a, and T are fixed, while Δt and θ are varied. In particular, $I = 1$, $a = 2$, $\Delta t = 1.25, 0.75, 0.5, 0.1$.

Figure 5: Report in PDF format generated from LaTeX source.

Sphinx. Sphinx[40] is a typesetting language with similarities to HTML and LaTeX, but with much less tagging. It has recently become very popular for software documentation and mathematical reports. Sphinx can utilize LaTeX for mathematical formulas and equations (via MathJax or PNG images). Unfortunately, the subset of LaTeX mathematics supported is less than in full MathJax (in particular, numbering of multiple equations in an `align` type environment is not supported). The Sphinx syntax[41] is an extension of the reStructuredText language. An attractive feature of Sphinx is its rich support for fancy layout of web pages[42]. In particular, Sphinx can easily be combined with various layout

[37] http://en.wikipedia.org/wiki/LaTeX
[38] http://tinyurl.com/k3sdbuv/writing_reports//_static/report.pdf
[39] http://tinyurl.com/k3sdbuv/writing_reports//_static/report.tex.html
[40] http://sphinx.pocoo.org/
[41] http://tinyurl.com/k3sdbuv/writing_reports//_static/report_sphinx.rst.html
[42] http://tinyurl.com/k3sdbuv/writing_reports//_static/sphinx-cloud/index.html

themes that give a certain look and feel to the web site and that offers contents, navigation, and search facilities, see Figure 6.

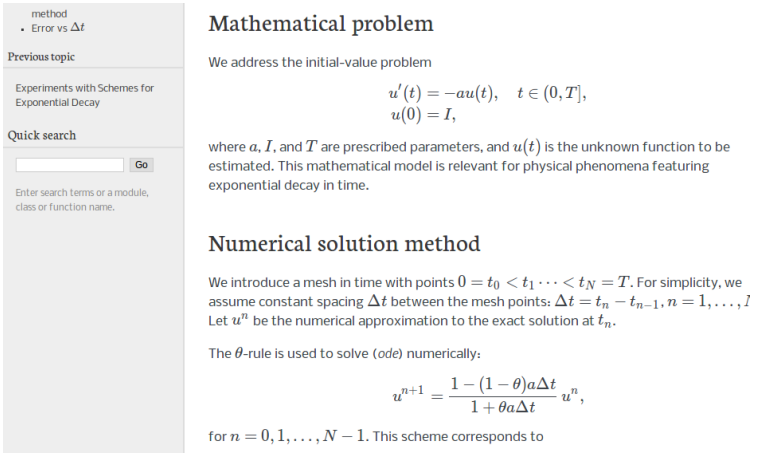


Figure 6: Report in HTML format generated from Sphinx source

Markdown. A recently popular format for easy writing of web pages is Markdown[43]. Text is written very much like one would do in email, using spaces special characters to naturally format the code instead of heavily tagged text as in LaTeX and HTML. With the tool Pandoc[44] one can go from Markdown to a variety of formats. HTML is a common output format, but LaTeX, XML, OpenOffice, MediaWiki, and MS Word are some other possibilities.

Wiki formats. A range of wiki formats are popular for creating content on the web, especially documents which allow groups of people to edit and share content. Apart from MediaWiki[45] (the wiki format used for Wikipedia), wiki formats have no support for mathematical typesetting and also limited support for displaying computer code in nice ways. Wiki formats are therefore less attractive for scientific reports compared to the other formats mentioned here.

DocOnce. Since it is difficult to choose the right tool or format for writing a scientific report, it is advantageous to write the content in a format that easily translates to LaTeX, HTML, Sphinx, Markdown, and various wikis. DocOnce[46] is such a tool. It is similar to Pandoc, but offers some special conveniences for writing about mathematics and programming. The tagging is more

[43] http://daringfireball.net/projects/markdown/
[44] http://johnmacfarlane.net/pandoc/
[45] http://www.mediawiki.org/wiki/MediaWiki
[46] https://github.com/hplgit/doconce
[47] http://tinyurl.com/k3sdbuv/writing_reports//_static/report.do.txt.html

omewhere between LaTeX and Markdown. The program `decay_exper_do.py` emonstrates how to generate (and write) DocOnce code for a report.

Worked example. The HTML, LaTeX (PDF), Sphinx, and DocOnce formats or the scientific report whose content is outlined above, are exemplified with ource codes and results at the web pages associated with this teaching material: `ttp://tinyurl.com/k3sdbuv/writing_reports/`.

.5 Publishing a complete project

report documenting scientific investigations should be accompanied by all the oftware and data used for the investigations so that others have a possibility to do the work and assess the qualify of the results. This possibility is important or *reproducible research* and hence reaching reliable scientific conclusions.

One way of documenting a complete project is to make a directory tree with ll relevant files. Preferably, the tree is published at some project hosting site like itbucket, GitHub, or Googlecode[48] so that others can download it as a tarfile, pfile, or clone the files directly using a version control system like Mercurial r Git. For the investigations outlined in Section 5.4, we can create a directory ee with files

```
setup.py
./src:
   decay_mod.py
./doc:
   ./src:
      decay_exper1_mathjax.py
      make_report.sh
      run.sh
   ./pub:
      report.html
```

he `src` directory holds source code (modules) to be reused in other projects, ie `setup.py` builds and installs such software, the `doc` directory contains the ocumentation, with `src` for the source of the documentation and `pub` for ready-ade, published documentation. The `run.sh` file is a simple Bash script listing ie `python` command we used to run `decay_exper1_mathjax.py` to generate ie experiments and the `report.html` file.

5 Exercises

Exercise 1: Refactor a flat program in terms of a function

or simple ODEs of the form

$$u' = f(t), \quad u(0) = I, \ t \in (0, T]$$

e can find the solution by straightforward integration:

[48] `http://hplgit.github.com/teamods/bitgit/html/`

$$u(t) = \int_0^t f(\tau)d\tau\,.$$

To compute $u(t)$ for $t \in [0, T]$, we introduce a uniform time mesh witl $t_n = n\Delta t$ and apply to Trapezoidal rule to approximate the integral. S we have computed the numerical approximation u^n to $u(t_n)$. We have

$$u(t_{n+1}) = u(t_n) + \int_{t_n}^{t_{n+1}} f(\tau)d\tau\,.$$

Using the Trapezoidal rule we get

$$u^{n+1} = u^n + \frac{1}{2}\Delta t(f(t_n) + f(t_{n+1}))\,.$$

The starting value is $u^0 = I$. A corresponding implementation for t $f(t) = 2t + 1$ is given next.

```
# f(t) is 2*t + 1
T = 2
from numpy import *
dt = 0.2               # time step
Nt = int(round(T/dt))  # no of mesh points
u = zeros(Nt+1)
t = linspace(0, T, Nt+1)
for n in range(Nt-1):
    u[n+1] = u[n] + 0.5*dt*(2*t[n]+1 + 2*t[n+1]+1)
```

This is a flat program. Refactor the program as a function `solver(f dt)`, where `f` is the Python implementation of the mathematical funct that is to be integrated. The return value of `solver` is the pair (`u`, `t`) repr the solution values and the associated time mesh. Filename: `integrat`

Remarks. Many prefer to do a first implementation of an algorithm program and hardcode formulas, here the $f(t)$, into the algorithm. Unfort this coding style makes it difficult to reuse a well-tested algorithm. the flat program works, it is strongly recommended to *refactor* the cc rearrange the statements) such that general algorithms are encapsul Python functions. The function arguments should be chosen such t function can be applied for a large class of problems, here all problems t be expressed as $u' = f(t)$,.

Exercise 2: Compare methods for a given time mesh

Make a program that imports the `solver` function from the `decay_mod` and offers a function `compare(dt, I, a)` for comparing, in a plot, the corresponding to $\theta = 0, 0.5, 1$ and the exact solution. This plot sh accuracy of the methods for a given time mesh. Read input data for the from the command line using appropriate functions in the `decay_mod`

he `-dt` option for giving several time step values can be reused: just use the first me step value for the computations). Filename: `decay_compare_theta.py`.

roblem 3: Write a doctest

ype in the following program and equip the `roots` function with a doctest:

```
import sys
# This sqrt(x) returns real if x>0 and complex if x<0
from numpy.lib.scimath import sqrt

def roots(a, b, c):
    """
    Return the roots of the quadratic polynomial
    p(x) = a*x**2 + b*x + c.

    The roots are real or complex objects.
    """
    q = b**2 - 4*a*c
    r1 = (-b + sqrt(q))/(2*a)
    r2 = (-b - sqrt(q))/(2*a)
    return r1, r2

a, b, c = [float(arg) for arg in sys.argv[1:]]
print roots(a, b, c)
```

Iake sure to test both real and complex roots. Write out numbers with 14 digits r less. Filename: `doctest_roots.py`.

roblem 4: Write a nose test

Iake a nose test for the `roots` function in Problem 3. Filename: `test_roots.py`.

roblem 5: Make a module

et

$$q(t) = \frac{RAe^{at}}{R + A(e^{at} - 1)}\,.$$

Iake a Python module `q_module` containing two functions `q(t)` and `dqdt(t)` or computing $q(t)$ and $q'(t)$, respectively. Perform a `from numpy import *` n this module. Import `q` and `dqdt` in another file using the "star import" onstruction `from q_module import *`. All objects available in this file is given y `dir()`. Print `dir()` and `len(dir())`. Then change the import of `numpy` n `q_module.py` to `import numpy as np`. What is the effect of this import on ne number of objects in `dir()` in a file that does `from q_module import *`? ilename: `q_module.py`.

xercise 6: Make use of a class implementation

Ve want to solve the exponential decay problem $u' = -au$, $u(0) = I$, for several t values and $\theta = 0, 0.5, 1$. For each Δt value, we want to make a plot where the

three solutions corresponding to $\theta = 0, 0.5, 1$ appear along with the exact s Write a function `experiment` to accomplish this. The function should the classes `Problem`, `Solver`, and `Visualizer` from the `decay_class`[49] and make use of these. A new command-line option `--dt_values` must b to allow the user to specify the Δt values on the command line (the opti and `-theta` implemented by the `decay_class` module have then no effe running the `experiment` function). Note that the classes in the `decay` module should *not* be modified. Filename: `decay_class_exper.py`.

Exercise 7: Generalize a class implementation

Consider the file `decay_class.py`[50] where the exponential decay proble $-au$, $u(0) = I$, is implemented via the classes `Problem`, `Solver`, and `Visu` Extend the classes to handle the more general problem

$$u'(t) = -a(t)u(t) + b(t), \quad u(0) = I,\ t \in (0, T],$$

using the θ-rule for discretization.

In the case with arbitrary functions $a(t)$ and $b(t)$ the problem cla longer guaranteed to provide an exact solution. Let the `u_exact` in class P return `None` if the exact solution for the particular problem is not a Modify classes `Solver` and `Visualizer` accordingly.

Add test functions `test_*()` for the nose testing tool in the modu add a demo example where the environment suddenly changes (modele abrupt change in the decay rate a):

$$a(t) = \begin{cases} 1, & 0 \leq t \leq t_p, \\ k, & t > t_p, \end{cases}$$

where t_p is the point of time the environment changes. Take t_p = make plots that illustrate the effect of having $k \gg 1$ and $k \ll 1$. Fi `decay_class2.py`.

Exercise 8: Generalize an advanced class implementa

Solve Exercise 7 by utilizing the class implementations in `decay_class_` Filename: `decay_class3.py`.

References

[1] H. P. Langtangen. *A Primer on Scientific Programming With Pytho* in Computational Science and Engineering. Springer, third edition,

[49] `http://tinyurl.com/jvzzcfn/decay/decay_class.py`
[50] `http://tinyurl.com/jvzzcfn/decay/decay_class.py`
[51] `http://tinyurl.com/jvzzcfn/decay/decay_class_oo.py`

ndex